

# Upper Body Tracking Using the Polhemus Fastrak

Scott McMillan  
Department of Computer Science  
Naval Postgraduate School  
Monterey, CA 93943

Technical Report NPSCS-96-002  
31 January 1996

## 1. Introduction

This technical report presents the design and implementation of the sensor system used to track the arms and rifle motions with respect to the upper body. This was performed as part of the AUSA '95 dismounted infantry work – an effort to insert humans into NPSNet's distributed virtual environment. The sensor work can be divided into three areas: (1) the Fastrak setup including sensor positions and interface to the computer as well as the device driver software, (2) the software interface to the Jack model and the inverse kinematics algorithm needed to achieve “reasonable” arm tracking, and (3) the implementation of the rifle tracking and the initial attempts at targeting.

In the next section, a description of the hardware is presented. This includes the serial cable for the Fastrak unit to the computer, transmitter and receiver (sensor) ports, and DIP switch settings. It also illustrates the coordinate systems associated with each of these hardware components. Section 3 presents the corresponding device driver that was developed.

In Section 4, the placement of the sensors on the person to be tracked is illustrated and the organization of the upper body tracking software is introduced. In particular, the top-level interface functions between the NPSNet application and the algorithms developed in this work are described. This section is followed by the inverse kinematics routines for arm tracking (Section 5), and the algorithms for rifle tracking and targeting (Section 6).

Following AUSA '95, development of the head-mounted display (HMD) interface to use the new Fastrak device driver was performed. Section 7 describes this work. The report concludes with a summary and discussion of the work that needs to be performed to make this a better system.

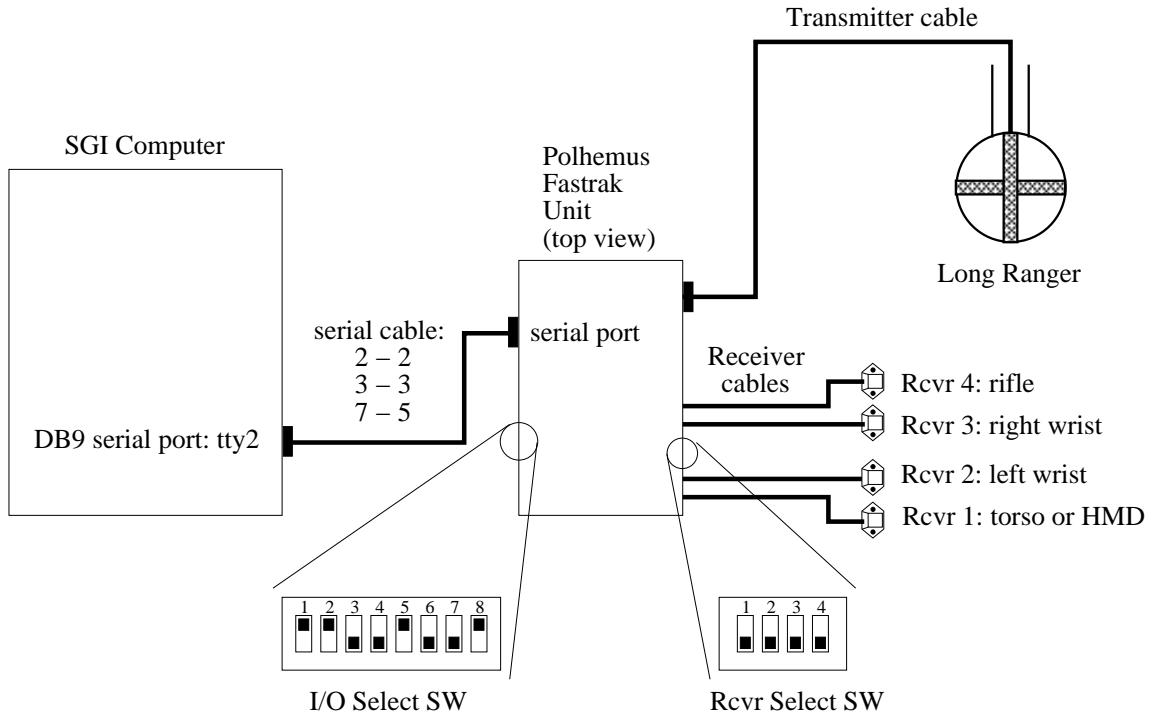


Figure 1: Fastrak hardware setup.

## 2. Polhemus Fastrak Hardware Setup

Figure 1 illustrates the hardware components used in the upper body sensor system. At the center is the Polhemus Fastrak unit. It is interfaced to the SGI workstation via a serial cable. In this case, a DB9 connector is assumed for the computer's serial port (as in the case of elvis or the Onyxes used at the AUSA'95 demonstration) the pins to be connected to support software flow control (i.e., XON/XOFF) are also given. Other cables for the DIN8 connector on Indigo2 workstations as well as hardware flow control cables can be found in the back of the Fastrak Reference Manual [1].

Also on the rear of the Fastrak unit, a set of eight DIP (I/O Select) switches are used to configure the unit's serial communication. The switch configuration shown in the figure specifies an RS232 9600 baud connection with 8 data bits and no parity, and enables software flow control (the configuration used at AUSA'95). Further documentation on these switches can be found on pages 11–14 of [1].

Note that although the Fastrak unit is capable of communicating at higher baud rates, the standard serial ports on most SGI workstations are incapable of exceeding 9600 baud using software flow control. Although attempts were made to use these higher baud rates with one of the specified hardware flow control cables, no success has been achieved in configuring the port properly in software to establish such a connection. This should be a topic for further investigation because the serial communication is a bottleneck to achiev-

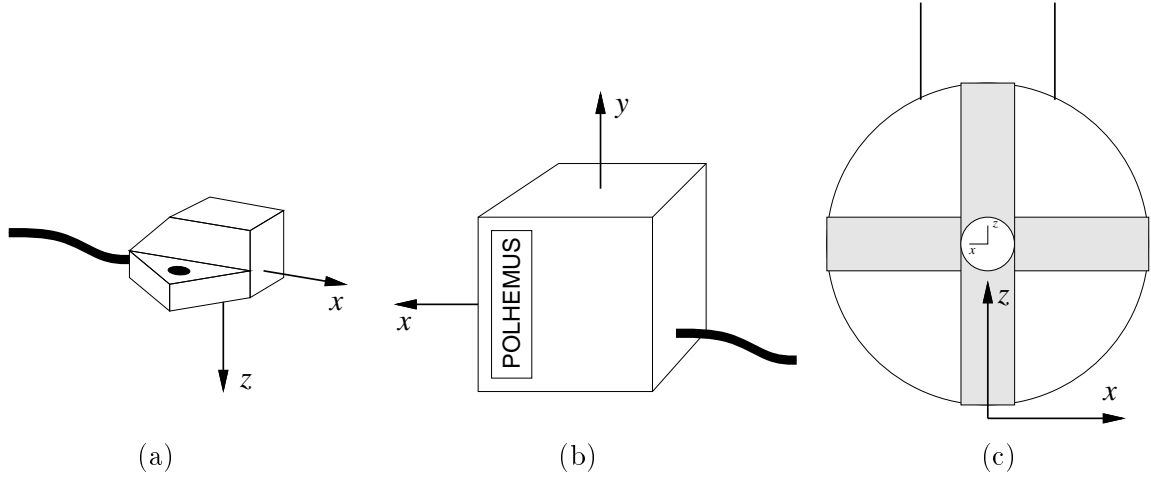


Figure 2: Coordinate systems for (a) a sensor (default), the standard transmitter (default), and (b) the Long Ranger (small - default; after setup - large).

ing sample rates higher than 15Hz when all four sensors are in use. Note that although frame update on the NPSNET application is 15Hz, higher baud rates would not only allow for communication of higher precision data but also permit filtering while introducing no additional lag.

On the front of the fastrak unit are the ports for the four sensors and the transmitter which can be either the short-range standard transmitter (the black box that comes with the unit) or the Long Ranger (also referred to as 'disco ball' which is currently mounted from the ceiling near elvis). Each sensor is referred to as a station in [1], and in order to ENABLE the unit process data for any station, the corresponding DIP switch on the front of the unit must be placed in the DOWN position.

Figures 2(b) and (c) give the default coordinate systems associated with the standard transmitter and the Long Ranger, respectively. The latter is on a small clear plastic "button" attached at the intersection of two of the coils as shown in the figure. The Fastrak unit provides the ability to redefine the transmitter's coordinate systems for each sensor, individually. in this application, however, the coordinate systems are expected to be the same for all sensors for the sake of simplicity.

The Long Ranger coordinate system used at the AUSA'95 show is given by the large arrows in Figure 2(c) with the  $x$ -axis out to the individual's right and the  $y$ -axis out the front in the direction of the individual's center of view. This is accomplished during initialization of the Fastrak unit described in the next section. This is also nearly equivalent to what is used to align the transmitter when using the HMD (see Section 7).

### 3. Device Driver: FastrakClass

The first task to be solved in this project was to develop the fastest possible interface to the Fastrak unit. This is built on a previous version of the device driver that was originally developed for the Isotrak at Sarcos and revised by Paul Barham and Jiang Zhu. Its purpose was to track one (HMD) and sometimes two (HMD and rifle) sensors. Although it was a very functional interface because it allowed the user to reconfigure the Fastrak on-the-fly, the approach resulted in very low sample rates which degraded further when tracking four sensors.

Realizing that most of the original functionality is unnecessary because once the unit is set up NPSNet will not change it, most of this functionality was discarded in an effort to gain speed. The result is a device driver that could allow configuration only during initialization, and sets up only one communication mode – a continuous binary stream of data – which was not supported by the previous version. This section discusses the initialization of the Fastrak unit, the communication setup implemented, the procedure for processing the data (the heart of the device driver), and the C++ class that was created to contain this driver along with the few accessor functions that have been provided as the user interface.

#### A. Polhemus Fastrak Initialization

The device driver and the Fastrak initialization are performed when the constructor of the `FastrakClass` is called:

```
FastrakClass(istream &config_fileobj,  
             short datatype_flags = FSTK_DEFAULT_MASK);
```

This is called from the `UpperBodyClass` constructor and must be passed an `istream` reference to the input configuration file (as shown in Appendix A) containing the parameters to configure the Fastrak unit. A second, optional parameter specifies what data types will be processed for all active sensors and sent over the serial connection. If omitted, the default data types are the three position coordinates and euler angles using a special 16BIT floating point format (see below).

The constructor inputs the data from the configuration file (`readConfig`), then it opens the IO port (`openIOPort`), configures the Fastrak unit, and spawns the process to handle the input (`initMultiprocessing`). The first parameter read from the configuration file is the device name of the port to which the unit is connected:

```
PORT: /dev/ttyd2
```

This is the second serial port, and the ‘d’ specifies that dumb terminal mode is desired. This mode requires only a three lead cable connecting RxD, TxD, and GND pins as illustrated in Figure 1. Other parameters read in from the file include the flags indicating which sensors (also referred to as stations) are to be active, and the hemisphere and alignment parameters for each sensor. In order to be valid, the active sensor flags must have corresponding DIP

switches in the front of the unit in the down position. For AUSA'95, all four sensors are active and all four switches are down resulting in the following line in the configuration file:

```
WANTED_STATIONS: 1 1 1 1
```

Note that the switches may be down for inactive sensors; however, the Fastrak unit appears to process its data anyway, but won't transmit it. This can lead to unnecessary delays.

The hemisphere and alignment parameters are described in [1] on pp. 88–91 and pp. 42–49 and are specified in the configuration file (in Appendix A) as follows:

```
STATIONx_PARAM:
    hemisphere: 0  0 -1
    origin:      0  0  0
    x_point:    -1  0  0
    y_point:      0 -1  0
```

In the first line, the hemisphere describes which half of the space around the transmitter the sensors are to be operated in. This is necessary since the computation by the Fastrak unit leads to two solutions for position and orientation and this information is needed to determine which is correct. Operating sensors outside this hemisphere leads to incorrect results. The hemisphere is defined relative to the transmitter's default coordinate system as shown in Figure 2. In this case, the hemisphere around the negative  $z$  axis will be used as the area of operation (which is the “bottom” half in its current configuration).

The other three lines define the alignment of the transmitter coordinate system with respect to its default one. The second line says to reference all position measurements with respect to the transmitters default origin. With the hemisphere defined as above, this origin appears to be at the bottom of the Long Ranger as shown in Figure 2. The last two lines effectively rotate the default coordinate system by 180 degrees about the  $z$ -axis so that  $x$  and  $y$  axes are pointing in their opposite directions. These hemisphere and alignment parameters can be defined differently for each sensor, but to maintain consistency required by the kinematics algorithms, it is the same for all sensors.

### *B. Communication Requirements with the Fastrak*

The previous version of the Fastrak device driver was written to explicitly poll the Fastrak unit for each packet of data. Using this approach, the 'P' command would be sent and the computer would wait for the Fastrak to sample the sensors sequentially, compute the solution, return the result in ASCII format over the serial communication line.

This approach suffered from a number of problems. First, the sample rate was determined by how fast the computer could send its requests which would always be less than the Fastrak was capable of producing because the process on the computer responsible for requesting the data would remain idle while the Fastrak was performing its sampling, computation and communication, and the Fastrak would be idle while the computer was processing this data. Although this was slightly alleviated by *sproc'ing* a process that would send requests as soon as it received a packet. The sample rate is still much lower than the maximum rates for the unit (120Hz with one sensor, 60Hz with two, and 30Hz with four).

Table 1: Maximum sample communication rates achievable with four Polhemus sensors versus baud rate, number of sensors and data format (assuming the transmission of three position values and three Euler angles). The (\*) indicates that 4 sensor rates are limited by the units ability to produce the data to 30Hz, and 2 sensor rates are limited to 60Hz.

	4 sensors@9600 baud	2 sensors@9600 baud 4 sensors@19.2K baud	1 sensor@9600 baud 2 sensors@19.2K baud 4 sensors@38.4K baud
ASCII	5.33Hz	10.67Hz	21.33Hz
IEEE-FP	8.89Hz	17.78Hz	35.56Hz(*)
16BIT	16Hz	32Hz(*)	64Hz(*)

In addition, as the number of sensors to be sampled increases the processing time increases because each sensor is processed sequentially (something the Ascension unit does not suffer from), and then the packet size to be transmitted increases and results in longer communication delays. Finally, the packet consisted of formatted floating point numbers represented by seven ASCII characters per number which is unnecessarily inefficient.

The first step was to configure the Fastrak unit to output a non-stop stream of data. In this mode, the unit starts the next sensor sampling period as soon as the previous one is completed. This removes the need to poll the unit for each packet and increases the Fastrak's sample rate. In this mode, a function is required to process the incoming data (see the subsequent discussions on the Serial Port Setup and Raw TTY Processing).

The next bottleneck to overcome was the speed of the serial communication of the data. Using the ASCII format, a minimum of 45 characters must be transmitted per sensor because the header consists of three characters and each of six floating point numbers (three for position and three Euler angles) is represented by seven characters. The maximum communication rate for these packets is listed in the first row of Table 1 for various baud rates and numbers of sensors. It shows that a communication rate of 38.4Kbaud is required to achieve the 15Hz sample rate required by the NPSNet application. This baud rate requires special hardware: either the Audio/Serial Option (ASO) for the Onyx (not available at AUSA'95), or a special hardware handshaking cable.

The latter was attempted without luck: either the cable specified in [1] was inadequate and/or the computer's serial port was configured incorrectly in software (no help could be obtained from Polhemus, Inc. because they are unfamiliar with SGI/Unix platforms). The alternative is to reduce the size of the packet to be transmitted. If single precision on IEEE floating point format is used, only four bytes per number are required which drops the packet size to 27 bytes per sensor and results in the rates presented in the second row of the table. Note that the 35.56Hz rate is achievable for the 1 and 2 sensor cases, whereas, the 4 sensor case is limited by the Fastrak unit's ability to produce samples at the 30Hz rate. However, the rate for four sensors at 9600 baud is still too low.

The solution to this problem is to use the Fastrak unit's special 16BIT floating point format (a 14 bit 2's-complement format with implicit ranges as specified on pp. 110–111

of [1]) that only requires two bytes per number. This reduces the packet size to 15 bytes per sensor will exceed the desired 15Hz sample rate while still using a simple (three-lead) serial cable at 9600 baud. Although this format suffers from reduced precision as compared to the IEEE floating point format, experience has shown that the electromagnetic noise affecting the sensor’s accuracy is high enough in normal operation to mask any increased quantization effects that might be seen with this format.

### C. Serial Port Setup

Using this mode, however, binary data will be transmitted by the Fastrak unit that would contain XON/XOFF characters that are not meant for flow control. Therefore, the computer’s serial port must be configured in a special way to prevent “hanging” when an XOFF character is transmitted. In addition there are no carriage returns or line feeds to indicate end-of-line, so that the standard canonical mode input processing cannot be used. Instead, the `termio` struct used to configure the required port is given as follows:

```
struct termio term;
memset(&term, 0, sizeof(term));

term.c_cflag = B9600|CS8|CLOCAL|CREAD|HUPCL;
term.c_iflag = IXOFF;
term.c_cc[VMIN] = 0;
term.c_cc[VTIME] = 5;
```

where the `c_cflag` configures the port for 8 data, 1 start, and 1 stop bits, 9600 baud and no parity.

The `c_iflag = IXOFF` allows software flow control in one direction so that the computer may suspend and resume the data flow using XON/XOFF characters while disabling flow control in the other direction to prevent the Fastrak data from affecting flow in the other direction. Non-canonical processing is specified using the `c_cc` field of the struct (see the `termio` man pages). With the `VMIN` parameter set to zero, a read on the port is satisfied as soon as a single character is received or a timer specified in tenths of seconds by the `VTIME` parameter expires. Note, however, that more than one character can be requested during any given `read` and up to that many characters can be received if the serial port’s buffer contains them.

### D. TTY Input Processing

Because the data is read in on a byte-by-byte basis, a routine is needed to query the port for data, take what data is available, identify which sensor it corresponds to, convert the data to IEEE floating point numbers and place them in buffers so that the NPSNet application can access them. The functions and buffers used to accomplish this task are illustrated in Figure 3.

After the serial port is configured as described above, a function called `pollContinuously` is spruced by the `FastrakClass` object. This function starts the Fastrak’s continuous

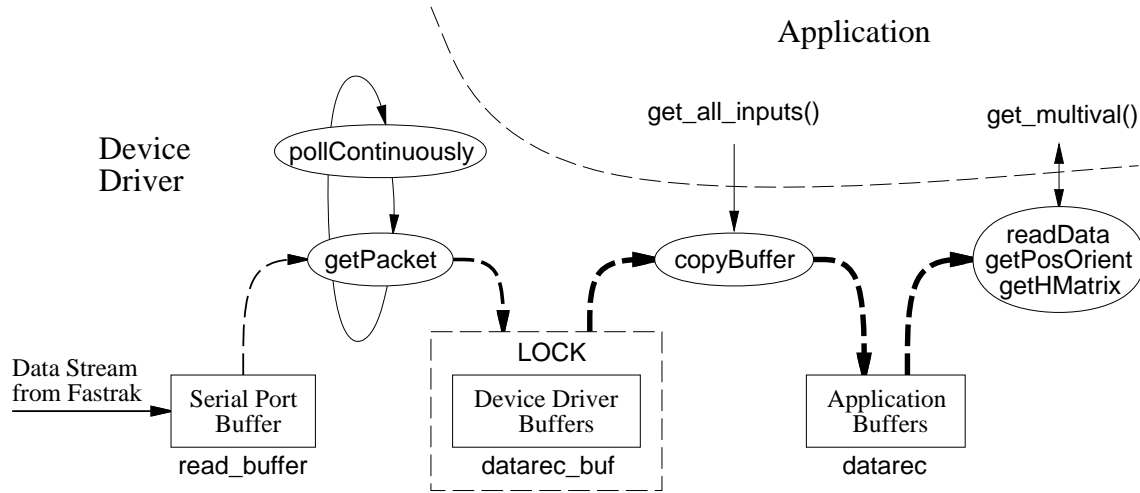


Figure 3: Software and buffer organization used in the device driver and application processes.

stream of data, calls `getPacket` (a member function of the `FastrakClass`) continuously until its parent process sends a quit signal, halts the stream of data, and exits. The `getPacket` function is a private member of the `FastrakClass` and performs all of the work required to convert the data stream into packets of data from each sensor. It reads data that is available from the serial port into a temporary character buffer called `read_buffer`. If no data is available, the read will time out as described above, and the `while` loop in `pollContinuously` will ensure the read is tried again.

If new data is read and there is enough to complete an entire packet for a single sensor, data will be processed and transferred to a second buffer. The processing consists of synchronizing with the stream, identifying which sensor the data corresponds to and transferring the appropriate number of bytes. The synchronization is accomplished by searching for a `0x303#` pattern in the three header bytes that precede each sensor’s data as follows:

0x30	0x3#	error code byte	data...
------	------	-----------------	---------

where “#” is the number corresponding the sensor’s number (i.e., `0x31` for the first sensor, `0x32` for the second sensor, etc.). If the procedure, is not already synchronized it will start discarding the data in `read_buffer` until this pattern is detected. When a complete packet for one sensor is collected, the data is transferred to the “device driver” buffer (`datarec_buf`).

The transfer to this second buffer is protected by a lock to ensure mutual exclusion. This will ensure that the application does not access this buffer while it is being updated. To reduce the accesses to this buffer a function called `copyBuffer` has been provided so that the application can copy the entire `datarec_buf` for all four sensors into yet another “application” buffer (`datarec`). For the NPSNet/Performer application, this function is called once at the beginning of each frame (approximately 15Hz) by calling the `UpperBodyClass`’s

member function called `get_all_inputs`. The data for all the sensors is transferred and then used throughout remaining computations.

### *E. Convenience Functions*

This implementation has been encapsulated within the C++ class, `FastrakClass`. Three public member functions of the `FastrakClass` are provided for the application to access, in a more “convenient” form, the data that the device driver has transferred to the `datarec` buffer. As shown in Figure 3 these are `readData`, `getPosOrient`, and `getHMatrix`. If necessary, these functions convert the internal 16BIT data format to IEEE single precision floating point numbers and place them in the appropriate arrays of floats. These functions are defined as follows:

```
int readData(FSTK_stations station_num, FSTK_datatypes data_type,
            float *data_dest);
```

`station_num` is an enumerated type with values (`FSTK_STATION1`, `FSTK_STATION2`, `FSTK_STATION3`, `FSTK_STATION4`) specifying which sensor’s data is to be accessed, `data_type` is an enumerated type that indicates which part of the sensor’s data is to be accessed, and `data_dest` points to an array floats where the data will be stored. In order to be valid, the data type requested must be one of the ones for which the Fastrak was configured to transmit to the computer during initialization. The possible values are as follows along with a description of what is returned in `data_dest`:

<code>FSTK_COORD_TYPE</code>	position data in centimeters or inches (3 floats)
<code>FSTK_EULER_TYPE</code>	Euler angles in degrees (3 floats)
<code>FSTK_XCOS_TYPE</code>	x-directional cosine (3 floats)
<code>FSTK_YCOS_TYPE</code>	y-directional cosine (3 floats)
<code>FSTK_ZCOS_TYPE</code>	z-directional cosine (3 floats)
<code>FSTK_QUAT_TYPE</code>	unit quaternion (4 floats)
<code>FSTK_16BIT_COORD_TYPE</code>	position data in centimeters or inches (3 floats)
<code>FSTK_16BIT_EULER_TYPE</code>	Euler angles in degrees (3 floats)
<code>FSTK_16BIT_QUAT_TYPE</code>	unit quaternion (4 floats)

Except for the last three, IEEE floating point data is assumed and a function to convert the data from the Fastrak’s byte-reversed format to the Unix format (`convertData`) is called before it is stored in the array. For the last three, the `convert16BITData` function is used to compute the floating point numbers. The function will return `TRUE` if no errors have been encountered.

```
int getPosOrient(FSTK_stations station_num, FSTK_datatypes orient_type,
                float pos[3], float *orient);
```

This provides a way of getting position and orientation in a single function call. The function determines which position format has been configured and calls the appropriate conversion routine and stores the result in an array of three floats pointed to by `pos`. If no position format has been specified during initialization, the function will return `FALSE`. The `orient_type` parameter can be any one of (`FSTK_EULER_TYPE`,

FSTK\_QUAT\_TYPE, FSTK\_16BIT\_EULER\_TYPE, FSTK\_16BIT\_QUAT\_TYPE) provided it was selected during initialization; otherwise, an error will result and the function will return `FALSE`. If a valid orientation type has been specified the result will be stored in the array pointed to by `orient` which must be either three or four (for quaternions) bytes long.

```
int getHMatrix(FSTK_stations station_num, float Hmatrix[4][4]);
```

This function computes a  $4 \times 4$  transformation matrix that describes the specified sensor's position and orientation relative to the transmitter,  ${}^{tx}\mathbf{H}_{rx}$ . The upper left  $3 \times 3$  portion of the matrix contains the rotation matrix. Depending on which orientation data type is specified during initialization, this is computed from the Euler angles or quaternions, or assigned from the directional cosine data that is provided from the data. If no orientation information has been specified, an error will occur and the function will return `FALSE`. If a position data type has been specified, the last column of the `Hmatrix` will be filled with this data; otherwise, it will be filled with zeroes. A 1 will be placed in the (4,4) position to complete the matrix.

**IMPORTANT NOTE:** this matrix is the transpose of the matrix that would be used by Performer functions that expect a `pfMatrix` with position data in the fourth row.

#### 4. NPSNet Interface: UpperBodyClass

To perform arm tracking, three of the four sensors are worn by the person to be tracked as shown in Figure 4. The wrist sensors are attached with velcro wrist bands aligning the sensor with the forearm so that the wire points along the forearm. Note that reasonable alignment is important as calibration does not measure the orientation of the sensor relative to the wrist. The torso sensor acts as a reference system and is worn high on the back on the outside of the shoulder harness with the sensor wire running down the back. The fourth sensor is mounted on the rifle. The preferred rifle sensor orientation is shown in the figure which aligns the sensor's axes as shown in Figure 2(a) with the graphical model of the rifles currently included with the Jack models ( $x$  along the barrel and  $z$  up).

##### A. Object-Oriented Design

The software was written in C++ using object-oriented design (OOD) concepts and was incorporated into NPSNet IV.9. The object and class hierarchies for this software are given in Figure 5(a) and (b), respectively. The top-level object, `upper_body`, is of type `UpperBodyClass` and provides the interface for the NPSNet software. The `upper_body` object also has four component objects consisting of the fastrak unit, two arms, and the rifle. The implementation of which is described in detail in the sections that follow.

To support the NPSNet interface, the `UpperBodyClass` inherits NPSNet's `input_device` class definition and implements all of the necessary virtual functions to support its use

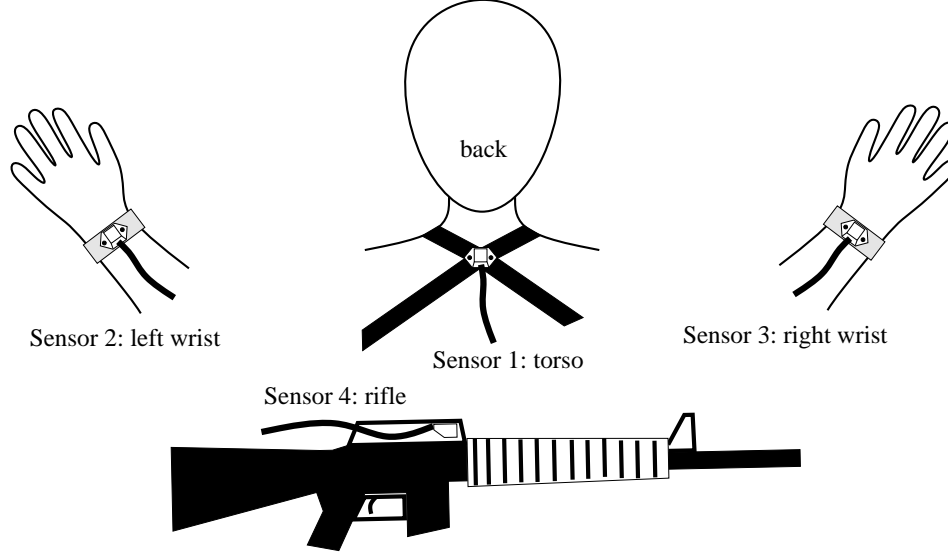


Figure 4: Polhemus sensor locations.

as an input device: `exists()`, `calibrate()`, `get_all_inputs()`, `set_multival()`, and `get_multival()`. A brief description of these interface functions are given as follows:

`UpperBodyClass(const char *cfg_filename)` The constructor, when called, results in the initialization of the entire system, including the creation of the other objects in the system and initialization of the Fastrak unit and serial communications. The single parameter provided is a string containing the full path and name of the file that specifies how the various elements of this system should be configured. An example file is listed in Appendix A.

`int exists()` This function returns `TRUE` (1) if everything was created and the Fastrak was initialized without any errors.

`int calibrate(const NPS_VALUATOR, const NPS_CALIBRATION prompt, const float)` This function must be called before the system is used to setup internal variables that are used in the kinematics computations for the arms and rifle. It currently initiates the arm/torso sensor calibration by having the users wearing the sensors place their arms in two calibration positions. Only the second argument, `prompt`, is used by this routine. If `prompt=NPS_UB_DELAY`, then a time delay will be used to sample the sensors at the appropriate times. In this mode, a three second delay will occur between the prompt printed on the screen and the actual sampling of the sensor. This is useful for when the user is alone and cannot press return. If `prompt=NPS_UB_PROMPT` or `prompt=NPS_CAL` is specified a second user will press return after the sensed user has placed their arms in each of the required positions. The operations performed in this calibration routine are discussed in the section describing the `ArmClasses`.

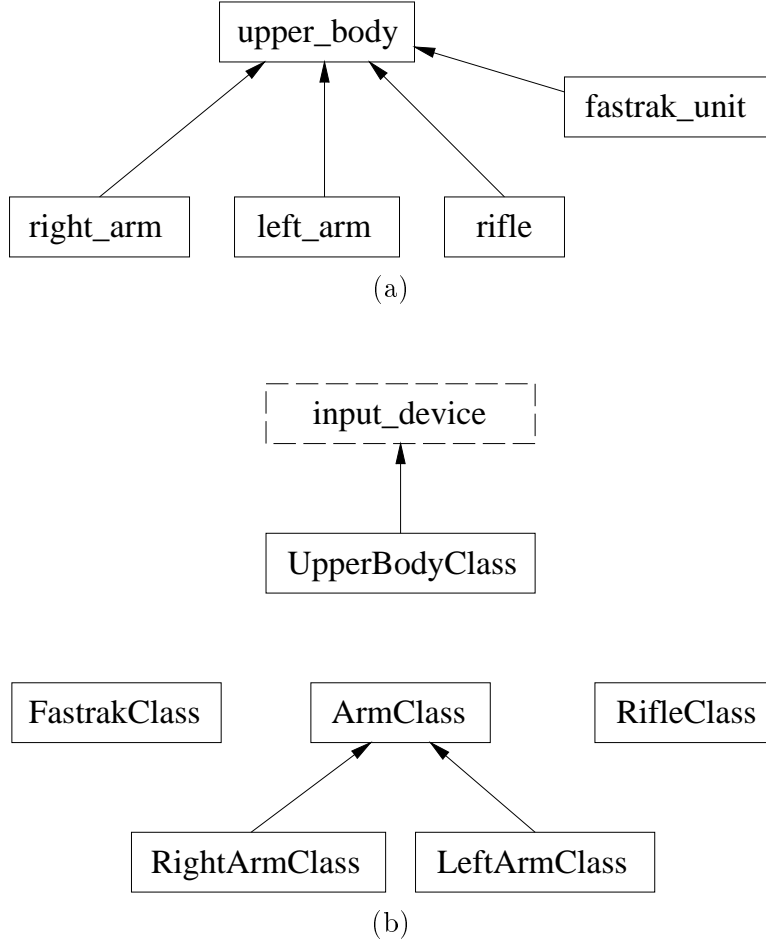


Figure 5: The object (a) and class (b) hierarchies for the software.

`void get_all_inputs(pfChannel *)` This function is used to transfer the latest data from the Fastrak device driver buffers to the member variables in the `upper_body` object and also scales the position data to be expressed in meters instead of centimeters. This is called from the NPSNet application at the beginning of each frame of computation, and is done to reduce the mutual exclusion overhead involved in accessing these buffers. See the section on `FastrakClass` for additional information on the device driver. The parameter is unused.

`int set_multival(const NPS_MULTIVAL tag, void *values)` This function is used for inputting data into the upper body software from the NPSNet application. This is currently used to set the transformation matrices that specify the position and orientation of both wrists with respect to the Jack model's `UPWAIST` coordinate system. The `tag` parameter is set to `NPS_MV_RIGHTHAND` to set the matrix for the right arm and `NPS_MV_LEFTHAND` to set the value for the left arm. In either case, `values` is a pointer to a `pfMatrix` which contains the transformation matrix as defined in the Performer

API (transposed with the position vector along the fourth row). This is done so that the rifle can be snapped to the hand positions (with reasonable accuracy) when the hands approach the appropriate positions. This is discussed in the section on the RifleClass.

`int get_multival(const NPS_MULTIVAL tag, void *values)` This function is used to initiate the various computations needed to perform the upper body and rifle tracking. If this function is called with `tag=NPS_MV_UPPERBODY`, then the inverse kinematics routines to compute the joint angles for both arms are executed. The results are returned in an array of `fixedDatumRecord` structs which is the `fixed_datum` element of a `DataPDU` struct that is pointed to by the `values` parameter. These structs are defined in `src/communication/include/pdu.h` as follows:

```
typedef struct {
    unsigned int    datum_id;
    float           datum;
} fixedDatumRecord;

typedef struct {
    PDUHeader        header;
    EntityID          orig_entity_id;
    ForceID           orig_force_id;
    EntityID          recv_entity_id;
    ForceID           recv_force_id;
    unsigned int      request_id;
    unsigned int      num_datum_fixed;
    fixedDatumRecord  fixed_datum[MAX_FIXED_DATUM_SIZE];
} DataPDU;
```

The enumerated type defined in `src/entities/include/jointmods.h` is used to index into this array which is defined as follows:

```
enum {
    LS0 = 0, LS1, LS2, LEO, LW0, LW1, LW2,    // Left arm joints
    RS0,    RS1, RS2, REO, RW0, RW1, RW2,    // Right arm joints
    RCX,    RCY, RCZ, RCH, RCP, RCR,          // Rifle pfCoord values
    NUM_HIRES_VALUES
};
```

The `get_multival` performs any conversions from internally defined joint angles to those needed for the Jack model and assigns them to the appropriate joint variables in this array. The conversions are listed in the last columns of Table 2 and a description of the inverse kinematics algorithms are contained in the section on `ArmClasses`.

If this function is called with `tag=NPS_MV_RIFLE`, then the function to compute the position and orientation of the rifle is computed and placed in the array of `fixedDatumRecords` pointed to by `values`.

If this function is called with `tag=NPS_MV_TARGET_RIFLE`, then a crude targeting function is executed and the result, the position and orientation of the vector describing the line of fire in the world coordinate system, is stored in a `pfCoord` that is pointed at by `values`. Both of these functions are described in more detail in the section on the `RifleClass`.

### B. Procedural Flow

This section describes the order of the top level calls to the functions listed in the previous section to perform arm and rifle tracking. This code is spread throughout the file `jack_di_veh.cc` (ask Randall Barker for implementation specifics):

1. **Initialize Jack:** besides the standard initialization, this also involves cloning the graphical rifle model removing it from the hull DCS and attaching it to Jack's UPWAIST DCS. Since all rifle tracking is performed with respect to the torso sensor, attaching the graphical model to the DCS which tracks with the upper torso simplifies some of the computation. Also set the joint overrides for the shoulders, elbows, and wrists.
2. **Initialize the upper body system:** Instantiate the object hierarchy by calling the `UpperBodyClass` constructor.
3. **Calibrate:** If construction is successful, call the `calibrate()` function.
4. **Main Loop:**
  - (a) **Get new data:** Update the buffers containing the latest Fastrak data by calling `get_all_inputs()`.
  - (b) **Compute arm joint angles:** This is accomplished with a call to `get_multival` with the tag set to `NPS_MV_UPPERBODY`.
  - (c) **Set the Jack model arm angles.**
  - (d) **Compute actual Jack wrist positions:** This computes the transformation matrices from Jack's UPWAIST coordinate system to each wrist. This accounts for model differences between Jack's and the one described in the next section for the inverse kinematics. Two calls to `set_multival` with the tag equal to `NPS_MV_RIGHTHAND` and `NPS_MV_LEFTHAND` are needed to transfer these results to the `UpperBodyClass` object.
  - (e) **Compute rifle position and orientation:** Using the position of the graphical wrists and the position and orientation of the real rifle (via sensor) data, determine the new rifle position. If the wrist positions are "close" to the corresponding grasping positions on the rifle, the rifle position and orientation are modified to "snap-to" the hands.
  - (f) **Repeat Main Loop.**

In the sections that follow, the algorithms to perform the calibration, arm joint angle computations, rifle position and orientation, and snap-to are described.

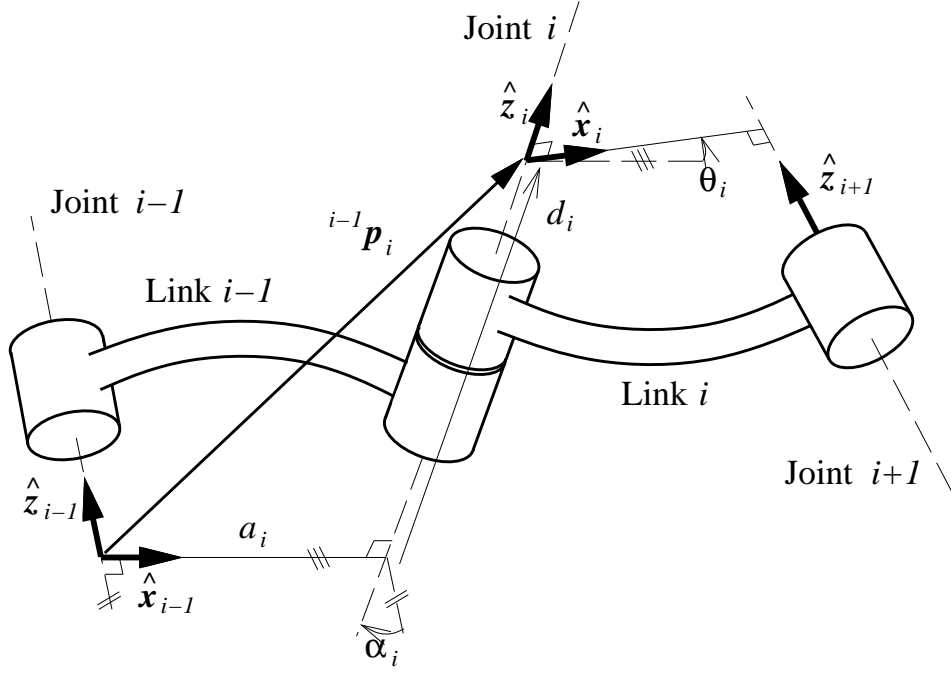


Figure 6: Link coordinate system assignment and MDH parameters.

## 5. Arm Tracking and the Jack Model: ArmClass

In this section, the `ArmClasses` are described. It contains the inverse kinematics code for each arm (individual functions are contained in the `RightArmClass` and `LeftArmClass`). These functions are based on the modified Denavit-Hartenberg (MDH) notation for specifying the kinematics of a linkage system; therefore, this notation is described first. Using this notation, the kinematics of both arms can be closely approximated as a series of seven links connected by revolute joints. Only the first five joints (three shoulder, one elbow, and the forearm roll) are considered in this work and presented in the second part of this section. Then the inverse kinematics algorithm is described. Finally, a brief description of the calibration routine is provided.

### A. Link Coordinate Systems and Modified Denavit-Hartenberg (MDH) Parameters

The most efficient dynamics algorithms for articulated mechanisms result when coordinate systems are assigned to each link in the system. For a vast majority of systems, the links in the serial chains are attached to one another with single degree-of-freedom revolute or prismatic joints. In these cases, it is efficient to assign these coordinate systems according to a set of rules similar to those first described by Denavit and Hartenberg in [2]. The resulting modified Denavit-Hartenberg (MDH) notation has also been presented by Craig in [3].

Using this notation, the links are numbered, in succession, from link 1, attached to the base, to the last link at the tip or end-effector, link  $N$ . As shown in Figure 6, the joint between links  $i - 1$  and  $i$  is labeled joint  $i$ , and the origin of coordinate frame  $i$  lies on the axis of this joint's motion. Further, this coordinate frame is fixed to link  $i$  at the end “nearest” the base.

The axes of each coordinate system are aligned so that the  $z$ -axis,  $\hat{z}$  lies along the axis of motion of the joint. This is the axis about which the joint rotates for revolute joints, and along which the joint translates for prismatic joints. The  $x$ -axis,  $\hat{x}$ , lies along the common normal between this joint axis and the next one in the chain as shown in Figure 6. With this arrangement, only four parameters are needed to describe the relative position and orientation between adjacent coordinate systems. To move from link  $i - 1$ 's coordinate system to link  $i$ , these parameters are defined as follows:

- $a_i$  = the perpendicular distance along  $\hat{x}_{i-1}$  from  $\hat{z}_{i-1}$  to  $\hat{z}_i$ ,
- $\alpha_i$  = the angle about  $\hat{x}_{i-1}$  from  $\hat{z}_{i-1}$  to  $\hat{z}_i$ ,
- $d_i$  = the perpendicular distance along  $\hat{z}_i$  from  $\hat{x}_{i-1}$  to  $\hat{x}_i$ , and
- $\theta_i$  = the angle about  $\hat{z}_i$  from  $\hat{x}_{i-1}$  to  $\hat{x}_i$ ,

where  $d_i$  is variable if joint  $i$  is prismatic, or  $\theta_i$  is variable if it is revolute. Note that when the base of the chain is fixed with respect to an inertial frame, it is convenient to place the origin of coordinate frame 0 coincident with coordinate system 1 with its  $z$ -axis also along joint 1. As a result, the only non-zero parameter is the joint variable,  $\theta_1$  or  $d_1$ .

These parameters are used to define the homogeneous transformation matrix (including the rotation matrix,  $\mathbf{R}$ , and position vector,  $\mathbf{p}$ ) from one coordinate frame to the next. This transformation from coordinate frame  $i - 1$  to frame  $i$  is defined as follows:

$${}^i\mathbf{H}_{i-1} = \left[ \begin{array}{c|c} {}^i\mathbf{R}_{i-1} & {}^i\mathbf{p}_{i-1} \\ \hline 0 & 1 \end{array} \right] = \left[ \begin{array}{ccc|c} c\theta_i & s\theta_i c\alpha_i & s\theta_i s\alpha_i & -a_i c\theta_i \\ -s\theta_i & c\theta_i c\alpha_i & c\theta_i s\alpha_i & a_i s\theta_i \\ 0 & -s\alpha_i & c\alpha_i & -d_i \\ \hline 0 & 0 & 0 & 1 \end{array} \right], \quad (1)$$

where  $s\alpha_i$ ,  $c\alpha_i$ ,  $s\theta_i$ , and  $c\theta_i$  denote  $\sin\alpha_i$ ,  $\cos\alpha_i$ ,  $\sin\theta_i$ , and  $\cos\theta_i$ , respectively. The inverse of this matrix specifies the transformation in the opposite direction and is given as follows:

$${}^{i-1}\mathbf{H}_i = \left[ \begin{array}{c|c} {}^i\mathbf{R}_{i-1}^T & -{}^i\mathbf{R}_{i-1}^T {}^i\mathbf{p}_{i-1} \\ \hline 0 & 1 \end{array} \right] = \left[ \begin{array}{ccc|c} c\theta_i & -s\theta_i & 0 & a_i \\ s\theta_i c\alpha_i & c\theta_i c\alpha_i & -s\alpha_i & -d_i s\alpha_i \\ s\theta_i s\alpha_i & c\theta_i s\alpha_i & c\alpha_i & d_i c\alpha_i \\ \hline 0 & 0 & 0 & 1 \end{array} \right]. \quad (2)$$

It is important to note these matrices are the transpose of the matrices used by the Performer API.

### B. The Jack Model

In this part, the models for both of Jack's arms are specified using the MDH notation from the previous section. Much of the background information for this comes from a technical report by Deepak Tolani from the University of Pennsylvania [4]. Each wrist sensor allows for up to six degrees of freedom (DOFs) to be tracked relative to the torso sensor.

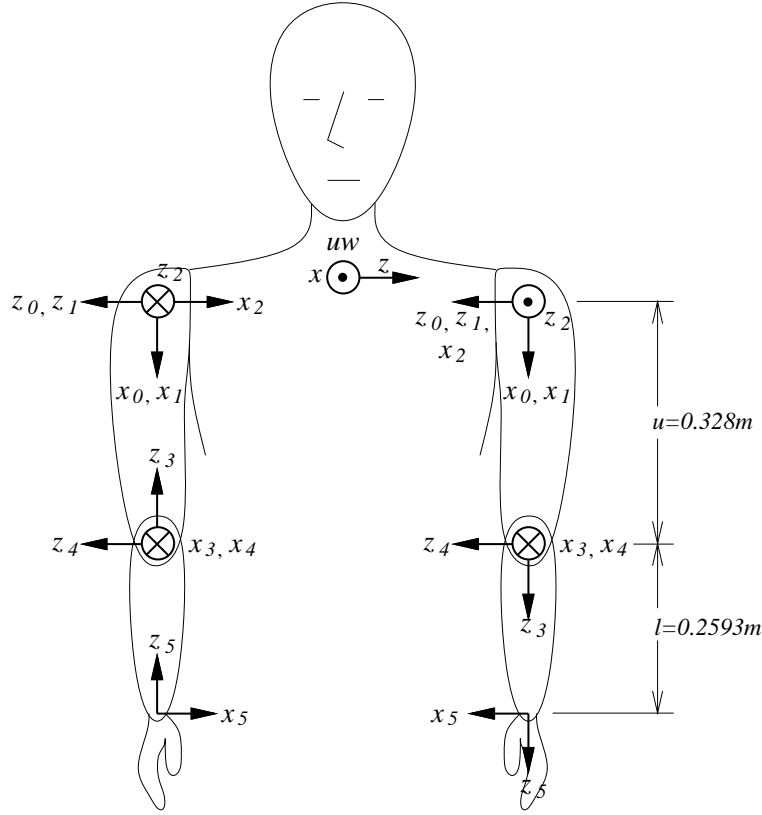


Figure 7: Jack's and MDH parameter coordinate systems.

With the clavicle joint fixed, only five DOFs are actually tracked: the three shoulder DOFs (it is approximated by a perfectly spherical ball joint), an elbow joint, and the forearm roll. Placing the sensor on the hand would have resulted in a seven DOF system which cannot be adequately tracked by a single sensor.

These five DOFs for each arm result in a set of six coordinate systems per arm as shown in Figure 7. The extra (zeroth) system acts as a base for each arm that is assumed to be fixed with respect to the torso sensor. Calibration, which is discussed later, is used to measure this constant offset. The upper and lower arm lengths have been specified by Tolani to be  $u = 0.328$  and  $l = 0.2593$  meters, respectively. Tolani also indicated a offset of the elbow joint of 0.5cm into the page when looking at Figure 7. This has been omitted to simplify the inverse kinematics computation without reducing the accuracy of the result, especially when sensor noise is considered.

The MDH parameters used in this work are listed in Table 2 for both arms. The arms in Figure 7 are shown in Jack's "zero" position where all joint angles are zero. This does not correspond to zero MDH joint angles,  $\theta_i$ , whose values are listed in fifth column of the tables. The last column of the table, therefore, specifies the conversions from MDH angle values to the corresponding Jack model angles along with the tokens used in the NPSNet software to index the proper joint angle variable. The MDH zero position puts both arms

Table 2: Modified Denavit-Hartenberg (MDH) parameters that correspond to the arm coordinate systems used by the Jack model including the simple transformations from MDH joint angles to Jack’s arm angles. The arm lengths, taken from Tolani [4], are  $u = 0.328$  and  $l = 0.2593$  meters.

Link $i$	$a_i$	$\alpha_i$	$d_i$	$\theta_i$	Jack angle
1	0	$0^\circ$	0	$0^\circ$	$\theta[\text{RS2}] = \theta_1$
2	0	$90^\circ$	0	$-90^\circ$	$\theta[\text{RS1}] = \theta_2 + 90^\circ$
3	0	$90^\circ$	$-u$	$90^\circ$	$\theta[\text{RS0}] = \theta_3 - 90^\circ$
4	0	$-90^\circ$	0	$0^\circ$	$\theta[\text{RE0}] = \theta_4$
5	0	$90^\circ$	$-l$	$-90^\circ$	$\theta[\text{RW2}] = \theta_5 + 90^\circ$

(a) Right arm.

Link $i$	$a_i$	$\alpha_i$	$d_i$	$\theta_i$	Jack angle
1	0	$0^\circ$	0	$0^\circ$	$\theta[\text{LS2}] = \theta_1$
2	0	$-90^\circ$	0	$-90^\circ$	$\theta[\text{LS1}] = \theta_2 + 90^\circ$
3	0	$-90^\circ$	$u$	$90^\circ$	$\theta[\text{LS0}] = \theta_3 - 90^\circ$
4	0	$90^\circ$	0	$0^\circ$	$\theta[\text{LE0}] = \theta_4$
5	0	$-90^\circ$	$l$	$-90^\circ$	$\theta[\text{LW2}] = \theta_5 + 90^\circ$

(b) Left arm.

straight out to the side with the palms down.

### C. Inverse Kinematics Algorithm

In this section, the inverse kinematics algorithm to compute the joint angles from the Fastrak sensor data is presented. Because both procedures are equivalent except for differences in parameters, only the inverse kinematics for the right arm is presented in this section. This corresponds to the function `RightArmClass::inverseKinematics5adj`.

#### Step 1. Sensor Input Transformation

The input required by the inverse kinematics algorithm is the homogeneous transformation matrix of the wrist’s coordinate system with respect to its shoulder’s,  ${}^{sh}\mathbf{H}_{rw}$  (or  ${}^0\mathbf{H}_5$  using the MDH coordinate system numbers). The sensor data obtained from the Fastrak device driver is the transformation of the sensors’ coordinate systems with respect to the transmitter’s *aligned* coordinate system.

To distinguish between the various sensors, the right wrist sensor’s transformation is referred to as  ${}^{tx}\mathbf{H}_{rws}$  and the torso sensor’s transformation is referred to as  ${}^{tx}\mathbf{H}_{ts}$ . To obtain the desired input, the following series of transformations must be performed:

$${}^{sh}\mathbf{H}_{rw} = {}^{sh}\mathbf{H}_{ts} {}^{tx}\mathbf{H}_{ts}^{-1} {}^{tx}\mathbf{H}_{rws} {}^{rws}\mathbf{H}_{rw}, \quad (3)$$

where  ${}^{sh}\mathbf{H}_{ts}$  is the transformation from the torso sensor the shoulder coordinate system

which is assumed to be constant because the clavicle joints are not modeled, and  ${}^{rw}s\mathbf{H}_{rw}$  is the constant transformation from the right wrist to the corresponding sensor's coordinate system. Both of these transformation matrices must be computed during the calibration that is described later.

*Step 2. Elbow angle,  $\theta_4$*

The distance between the shoulder and the wrist is given by the magnitude of the position vector,  ${}^{sh}\mathbf{p}_{rw}$ , taken from the result of Eq. (3). Using the rule that  $\|{}^{sh}\mathbf{p}_{rw}\| = \|{}^{rw}\mathbf{p}_{sh}\|$ , a mathematical expression for the corresponding distance in the graphical model is computed from the position vector,  ${}^5\mathbf{p}_0$ , in the following homogeneous transformation:

$$\left[ \begin{array}{c|c} {}^5\mathbf{R}_0 & {}^5\mathbf{p}_0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] = {}^5\mathbf{H}_4 {}^4\mathbf{H}_3 {}^3\mathbf{H}_2 {}^2\mathbf{H}_1 {}^1\mathbf{H}_0, \quad \text{where} \quad (4)$$

$${}^5\mathbf{p}_0 = \begin{bmatrix} -us_4c_5 \\ us_4s_5 \\ l + uc_4 \end{bmatrix}. \quad (5)$$

The magnitude of this distance simplifies to a function of only  $\theta_4$ , the elbow joint angle, as follows:

$$\|{}^5\mathbf{p}_0\|^2 = u^2 + l^2 + 2ulc_4 \quad (6)$$

If the real arm is within the graphical arms reachable volume the elbow joint angle can be computed normally as follows:

$$\theta_4 = \arccos\left(\frac{\|{}^{sh}\mathbf{p}_{rw}\|^2 - u^2 - l^2}{2ul}\right) \quad (7)$$

This ensures that the graphical model's wrist is as far away from the shoulder as the real one.

If  $\|{}^{sh}\mathbf{p}_{rw}\| > (u + l)$ , the wrist sensor has gone farther from the shoulder than the graphical model is capable of reaching. In this case,  $\theta_4$  is set to  $0^\circ$  because a workspace boundary has been exceeded and the elbow joint is fully extended to get closest to desired position. If  $\|{}^{sh}\mathbf{p}_{rw}\| < (u - l)$ , the graphical wrist is trying to be driven too close to the shoulder (the other workspace boundary) and  $\theta_4$  is set to  $180^\circ$ . Finally, the term inside of the parenthesis of Eq. (7) should not stray outside the range  $[-1, 1]$  due to roundoff or truncation errors before performing the arccos function.

*Step 3. Forearm roll angle,  $\theta_5$*

The next step is to compute the forearm roll angle from the position vectors,  ${}^{sh}\mathbf{p}_{rw}$  and  ${}^5\mathbf{p}_0$ . First, the position of the shoulder with respect to the sensed position of the wrist is computed using the terms in Eq. (3):

$${}^{rw}\mathbf{p}_{sh} = -{}^{sh}\mathbf{R}_{rw}^T {}^{sh}\mathbf{p}_{rw}, \quad (8)$$

which corresponds to  ${}^5\mathbf{p}_0$  in the graphical model:

$${}^{rw}\mathbf{p}_{sh} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} -us_4c_5 \\ us_4s_5 \\ l + uc_4 \end{bmatrix}. \quad (9)$$

With  $\theta_4$  computed from the previous step, the first two elements of  ${}^{rw}\mathbf{p}_{sh}$  are used to compute  $\theta_5$ :

$$\theta_5 = \arctan\left(\frac{p_y/us_4}{-p_x/us_4}\right). \quad (10)$$

Note, however, that if  $\sin(\theta_4) = 0$  (or close), the two elements of the position vector used here will be zero (or close). and  $\theta_5$  is undefined (or poorly defined due to roundoff errors). This is one of the singularities in the arm where the number of degrees of freedom drops to four which occurs when the arm is fully extended or the elbow is bent  $180^\circ$ . Although the latter is not physically possible, the situation can occur periodically with sensor and calibration noise that is present. In this case, the value of  $\theta_5$  does not matter, so it is set to its most recent previous valid value to prevent discontinuous jumps in the graphical model.

*Step 4. Shoulder angles,  $\theta_{1,2,3}$*

With  $\theta_4$  and  $\theta_5$  computed, the last step is to compute the three shoulder angles from the orientation information. To do this the sensed orientation of the shoulder joints is equated to the mathematical expression for the graphical model. The sensed orientation is computed with the following rotation matrix:

$${}^0\mathbf{R}_3^{sensed} = {}^{sh}\mathbf{R}_{rw} {}^5\mathbf{R}_4(\theta_5) {}^4\mathbf{R}_3(\theta_4) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}, \quad (11)$$

where the definition for  ${}^i\mathbf{R}_{i-1}$  is defined in Eq (1). The mathematical expression for the corresponding matrix in the graphical model is computed as follows:

$${}^0\mathbf{R}_3 = {}^0\mathbf{R}_1 {}^1\mathbf{R}_2 {}^2\mathbf{R}_3 = \begin{bmatrix} c_1c_2c_3 + s_1s_3 & -c_1c_2s_3 + s_1c_3 & c_1s_2 \\ s_1c_2c_3 - c_1s_3 & -s_1c_2s_3 - s_1c_3 & s_1s_2 \\ s_2c_3 & -s_2s_3 & -c_2 \end{bmatrix}. \quad (12)$$

The procedure for computing the three shoulder angles comes from equating various elements of the matrices in Eqs. (11) and (12). The second angle is computed first using one of two equivalent functions:

$$\theta_2 = \arctan\left(\frac{\sqrt{r_{31}^2 + r_{32}^2}}{-r_{33}}\right) = \arctan\left(\frac{\sqrt{r_{13}^2 + r_{23}^2}}{-r_{33}}\right) \quad (13)$$

If  $\sin\theta_2 \neq 0^\circ$ , then the computation of the final two angles is straightforward:

$$\theta_1 = \arctan\left(\frac{-r_{23}/\sin\theta_2}{r_{13}/\sin\theta_2}\right), \quad (14)$$

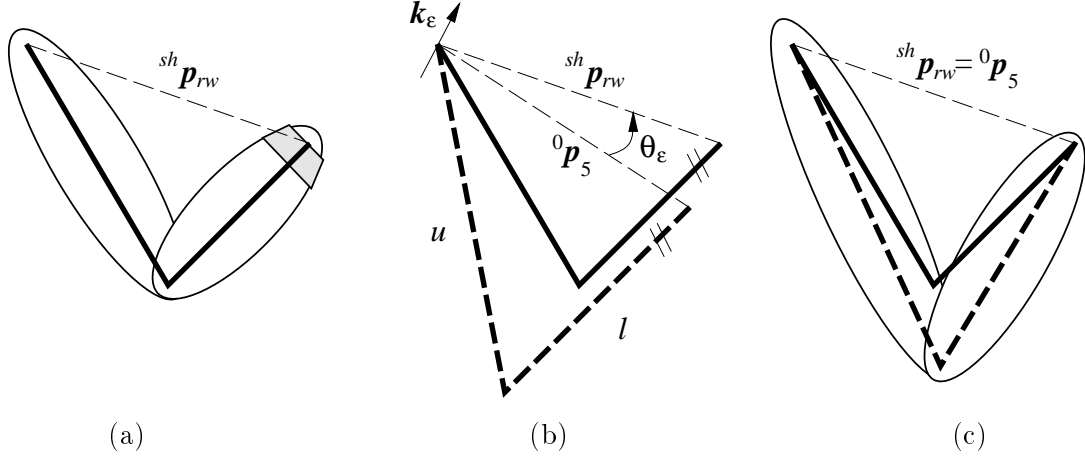


Figure 8: Steps in inverse kinematics procedure to drive Jack's arms to the desired position: (a) use original inverse kinematics to get proper orientation (Steps 2, 3, and 4), (b) determine axis and angle to rotate arm to proper position (Step 5), and (c) update shoulder angles to affect this rotation (Step 6).

$$\theta_3 = \arctan \left( \frac{-r_{32}/\sin \theta_2}{r_{31}/\sin \theta_2} \right). \quad (15)$$

If  $\sin(\theta_2) = 0$ , the second type of singularity has occurred in which the first and third joint axes line up and another degree of freedom is lost, and the previous equations to compute  $\theta_1$  and  $\theta_3$  cannot be used. Instead, only a linear combination of these angles can be computed. This is accomplished by examining the (1,1) and (2,1) elements of the rotation matrices and using the fact that  $\sin(\theta_2) = 0$  and  $\cos(\theta_2) = 1$ :

$$r_{11} = c_1 c_3 + s_1 s_3 = \cos(\theta_1 - \theta_3), \quad (16)$$

$$r_{21} = s_1 c_3 - c_1 s_3 = \sin(\theta_1 - \theta_3), \quad \text{and} \quad (17)$$

$$(\theta_1 - \theta_3) = \arctan \left( \frac{r_{21}}{r_{11}} \right). \quad (18)$$

The new value for  $\theta_3$  is then set to its previous value and  $\theta_1(new) = (\theta_1 - \theta_3) + \theta_3(new)$ .

*Step 5. Position error,  $(\mathbf{k}_\epsilon, \theta_\epsilon)$*

If the real and graphical arms were the same size and no noise was present in the sensor readings, the joint angles computed in the previous steps would place the graphical arm in the same position and orientation as the real arm. This is not the case as is illustrated in Figure 8. Figure 8(a) shows an example of a real arm configuration that has not exceeded the graphical arm's workspace. In Figure 8(b), the graphical arm configuration computed by Steps 2–4 are indicated by the dashed lines. The result of these steps ensures that the orientation of the wrist will be the same for both arms; however, the position of the wrists will be different because the upper and lower arm lengths are different.

Since a goal is to have the arms snap-to the rifle (or vice versa), it becomes more important to drive the graphical arms to the proper position at the expense of some error in the final orientation. This will reduce amount of discontinuity when snap-to occurs. Therefore additional steps are required to modify the results of the previous step.

The first step is illustrated in Figure 8(b) and involves the identification of the axis and angle of rotation  $(\mathbf{k}_\epsilon, \theta_\epsilon)$  that is required to rotate the graphical arm so that  ${}^0\mathbf{p}_5$  lines up with  ${}^{sh}\mathbf{p}_{rw}$ . The latter is determined from the sensor data in Step 1, and the former must be determined by computing the complete forward kinematics using the joint angles computed in Steps 2–4:

$$\left[ \begin{array}{c|c} {}^0\mathbf{R}_5 & {}^0\mathbf{p}_5 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] = {}^0\mathbf{H}_1(\theta_1) {}^1\mathbf{H}_2(\theta_2) {}^2\mathbf{H}_3(\theta_3) {}^3\mathbf{H}_4(\theta_4) {}^4\mathbf{H}_5(\theta_5). \quad (19)$$

The normalized axis is computed from the cross-product as follows:

$$\mathbf{k}_\epsilon = \frac{{}^0\mathbf{p}_5 \times {}^{sh}\mathbf{p}_{rw}}{\|{}^0\mathbf{p}_5 \times {}^{sh}\mathbf{p}_{rw}\|}, \quad (20)$$

and the angle can be computed from the following [5]:

$$\theta_\epsilon = \arctan \left( \frac{\|{}^0\mathbf{p}_5 \times {}^{sh}\mathbf{p}_{rw}\|}{{}^0\mathbf{p}_5 \cdot {}^{sh}\mathbf{p}_{rw}} \right). \quad (21)$$

Finally, the rotation matrix that corresponds to this angle and axis rotation,  $\mathbf{R}(\mathbf{k}_\epsilon, \theta_\epsilon)$ , must be computed. Note that when using Performer, the previous two equations can be replaced with a call to `pfMakeRotOntoMat`( $\mathbf{R}(\mathbf{k}_\epsilon, \theta_\epsilon)$ ,  ${}^{sh}\mathbf{p}_{rw}$ ,  ${}^0\mathbf{p}_5$ ) which computes the matrix directly.

*Step 6. Recompute shoulder angles,  $\theta_{1,2,3}$*

Since there are three angles about non-collinear axes (usually) at the shoulder, only the shoulder angles need to be modified to achieve the desired rotation computed in Step 5. This is accomplished by modifying  ${}^0\mathbf{R}_3^{sensed}$  in Eq. (11) as follows:

$${}^0\mathbf{R}_3^{sensed} = \mathbf{R}(\mathbf{k}_\epsilon, \theta_\epsilon) {}^{sh}\mathbf{R}_{rw} {}^5\mathbf{R}_4(\theta_5) {}^4\mathbf{R}_3(\theta_4). \quad (22)$$

With this matrix, the computation of  $\theta_{1,2,3}$  in Step 4 is repeated. The result will be a new set of shoulder angles that will position the wrist of the graphical model the same as the real one as illustrated in Figure 8(c). Since, the angle and axis approach is used, which defines the “shortest path” to the proper position, the resulting orientation of the wrist will be as close as possible to the original orientation.

#### D. Calibration

As stated before, the purpose of calibration is to determine the constant transformation matrices from the wrist sensors to the model’s wrist coordinate systems ( ${}^{lws}\mathbf{H}_{lw}$  and  ${}^{rws}\mathbf{H}_{rw}$ ), and from the model’s shoulder coordinate systems and the torso sensor ( ${}^{lsh}\mathbf{H}_{ts}$  and  ${}^{rsh}\mathbf{H}_{ts}$ ) that are used in Step 1 of the inverse kinematics algorithm. There is currently

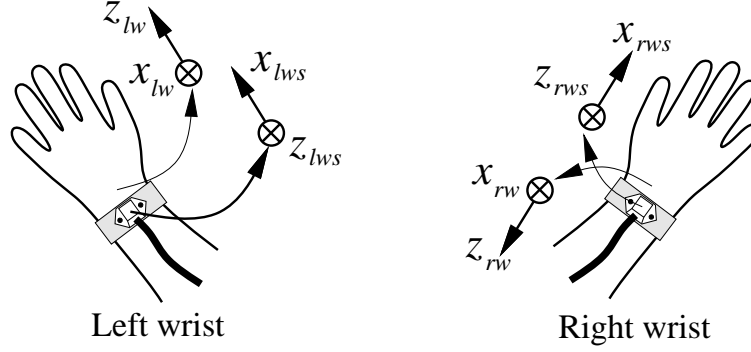


Figure 9: Wrist sensor calibration to wrist coordinate systems.

no way to measure the position and orientation of the wrist sensors with respect to the corresponding wrist coordinate systems for the model. Therefore, it is assumed that the sensors will be worn in a very specific orientation with respect to the wrists as shown in Figure 9 where  $rw$  and  $lw$  correspond to coordinate system 5 in Figure 7 for right and left arms, respectively.

Currently, the origin of the sensor coordinate system is assumed to be at the same point as the origin of the wrist coordinate system. This introduces an error of a few centimeters (not more than the noise inherent in the sensor data). It is also assumed that the  $x$  axis of the sensor ( $lws$  or  $rws$ ) is aligned with the forearms. This results in the following calibration matrices for both wrists:

$${}^{lws}\mathbf{H}_{lw} = \left[ \begin{array}{ccc|c} 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right], \quad (23)$$

$${}^{rws}\mathbf{H}_{rw} = \left[ \begin{array}{ccc|c} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right]. \quad (24)$$

If the offset of the origins were desired it could be added to the last column of both matrices.

Determining the transformation from the torso sensor to shoulder coordinate systems requires data from the wrist sensors after placing the arms in two different calibration positions as shown in Figure 10. First, the positions of the shoulder coordinate systems with respect to the transmitter,  ${}^{tx}\mathbf{p}_{rsh}$  and  ${}^{tx}\mathbf{p}_{lsh}$ , are determined:

$${}^{tx}\mathbf{p}_{rsh} = \frac{1}{2} \left( {}^{tx}\mathbf{p}_{rws}^{side} + {}^{tx}\mathbf{p}_{lws}^{side} \right) + \frac{1}{2} \left( {}^{tx}\mathbf{p}_{rws}^{front} - {}^{tx}\mathbf{p}_{lws}^{front} \right), \quad (25)$$

$${}^{tx}\mathbf{p}_{lsh} = \frac{1}{2} \left( {}^{tx}\mathbf{p}_{rws}^{side} + {}^{tx}\mathbf{p}_{lws}^{side} \right) - \frac{1}{2} \left( {}^{tx}\mathbf{p}_{rws}^{front} - {}^{tx}\mathbf{p}_{lws}^{front} \right), \quad (26)$$

where the *side* measurements have been used to determine the point halfway between

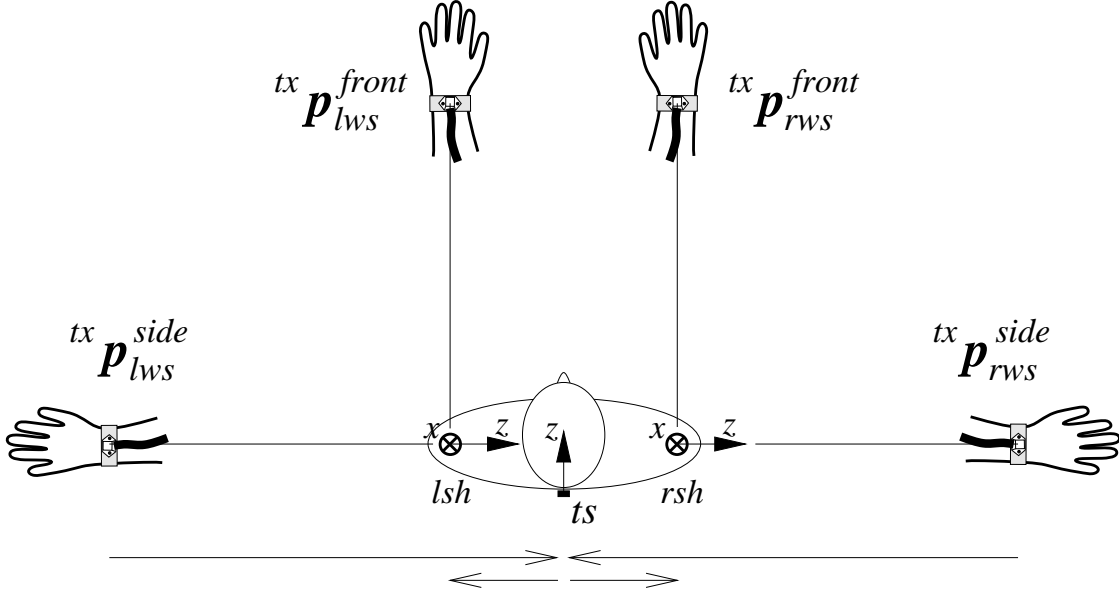


Figure 10: Torso sensor calibration to shoulder coordinate systems.

the shoulders, and the front measurements have been used to determine the width of the shoulders, and hence, the offset from the center point.

Second, the orientation of the shoulder coordinate systems with respect to the transmitter,  ${}^{tx}\mathbf{R}_{rsh}$  and  ${}^{tx}\mathbf{R}_{lsh}$ , are estimated. This procedure begins with the assumption that the  $z$  axis of the *aligned* transmitter points “up” and is reasonably plumb. With the  $x$  axes of both shoulder coordinate systems pointing down with the person stands up straight then the  $x$  axis of the shoulder coordinate systems with respect to the transmitter are given as:

$${}^{tx}\hat{\mathbf{x}}_{sh} = [0 \ 0 \ -1]^T. \quad (27)$$

In either calibration position, the  $z$  axis is parallel to the difference in the positions of the right and left wrist sensors:

$${}^{tx}\hat{\mathbf{z}}_{sh} = \frac{{}^{tx}\mathbf{p}_{rws}^{side} - {}^{tx}\mathbf{p}_{lws}^{side}}{\|{}^{tx}\mathbf{p}_{rws}^{side} - {}^{tx}\mathbf{p}_{lws}^{side}\|}. \quad (28)$$

The  $y$  axis is assumed to be perpendicular to these:

$${}^{tx}\hat{\mathbf{y}}_{sh} = \frac{{}^{tx}\hat{\mathbf{z}}_{sh} \times {}^{tx}\hat{\mathbf{x}}_{sh}}{\|{}^{tx}\hat{\mathbf{z}}_{sh} \times {}^{tx}\hat{\mathbf{x}}_{sh}\|}. \quad (29)$$

Finally, the  $z$  axis is adjusted to ensure the rotation matrix to be computed from this is orthogonal:

$${}^{tx}\hat{\mathbf{z}}_{sh} = \frac{{}^{tx}\hat{\mathbf{x}}_{sh} \times {}^{tx}\hat{\mathbf{y}}_{sh}}{\|{}^{tx}\hat{\mathbf{x}}_{sh} \times {}^{tx}\hat{\mathbf{y}}_{sh}\|}. \quad (30)$$

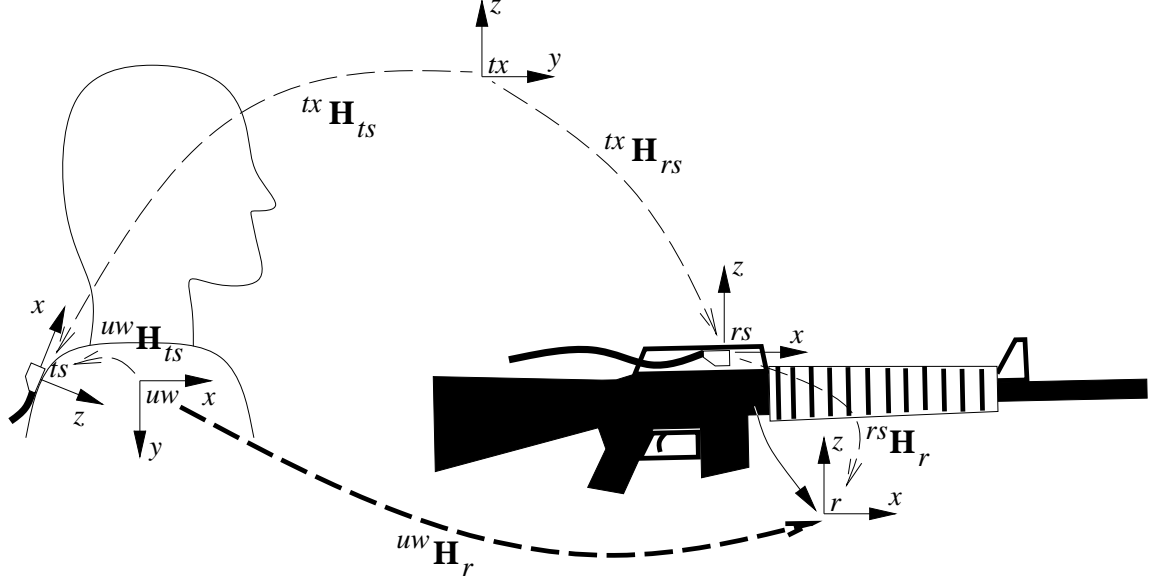


Figure 11: Coordinate systems used in rifle tracking.

From these results the desired rotation matrices are obtained as follows:

$${}^{tx}\mathbf{R}_{rsh} = {}^{tx}\mathbf{R}_{lsh} = \begin{bmatrix} {}^{tx}\hat{\mathbf{x}}_{sh} & {}^{tx}\hat{\mathbf{y}}_{sh} & {}^{tx}\hat{\mathbf{z}}_{sh} \end{bmatrix}. \quad (31)$$

Finally, the desired calibration matrices can be computed:

$${}^{rsh}\mathbf{H}_{ts} = \left[ \begin{array}{c|c} {}^{tx}\mathbf{R}_{rsh} & {}^{tx}\mathbf{p}_{rsh} \\ \hline 0 & 1 \end{array} \right]^{-1} {}^{tx}\mathbf{H}_{ts}^{side} \quad \text{and} \quad (32)$$

$${}^{lsh}\mathbf{H}_{ts} = \left[ \begin{array}{c|c} {}^{tx}\mathbf{R}_{lsh} & {}^{tx}\mathbf{p}_{lsh} \\ \hline 0 & 1 \end{array} \right]^{-1} {}^{tx}\mathbf{H}_{ts}^{side}, \quad (33)$$

where  ${}^{tx}\mathbf{H}_{ts}^{side}$  is the transformation matrix describing the position and orientation of the torso sensor with respect to the transmitter and is input from the Fastrak device driver during calibration at the same time the arms are out to the side.

## 6. Rifle Tracking: RifleClass

Since the rifle is a single rigid body, tracking is much simpler. A sensor is attached to the rifle and its position and orientation is tracked with respect to the torso sensor. To make computation simpler the new graphical model for the rifle is attached to Jack's UPWAIST coordinate system which is indicated by the  $uw$  axes in Figure 7 and again in Figure 11.

### A. Tracking and Calibration

In animating the graphical rifle, its position and orientation with respect to Jack's UPWAIST coordinate system must be determined as specified by the homogeneous transformation matrix,  ${}^{uw}\mathbf{H}_r$ , in Figure 11. This requires the sensor information for both the torso and rifle sensors,  ${}^{tx}\mathbf{H}_{ts}$  and  ${}^{tx}\mathbf{H}_{rs}$ . The formula to determine this matrix is given as follows:

$${}^{uw}\mathbf{H}_r = {}^{uw}\mathbf{H}_{ts} {}^{tx}\mathbf{H}_{ts}^{-1} {}^{tx}\mathbf{H}_{rs} {}^{rs}\mathbf{H}_r, \quad (34)$$

where  ${}^{rs}\mathbf{H}_r$  and  ${}^{uw}\mathbf{H}_{ts}$  are two calibration matrices that must be determined before tracking begins. Once this matrix is obtained the equivalent `pfCoord` can be computed with a call to Performer's `pfGetOrthoMatCoord`.

The matrix,  ${}^{rs}\mathbf{H}_r$ , describes the position and orientation of the graphical rifle's coordinate system with respect to the real one's sensor. This is a constant matrix since the sensor is rigidly attached to the rifle that is held by the user. The sensor position and orientation as illustrated in Figure 11 is determined off-line and given as follows:

$${}^{rs}\mathbf{H}_r = \left[ \begin{array}{ccc|c} 1 & 0 & 0 & 0.05 \\ 0 & 1 & 0 & 0.0 \\ 0 & 0 & 1 & -0.10 \\ \hline 0 & 0 & 0 & 1 \end{array} \right], \quad (35)$$

where the rotation portion is an identity matrix because the axes of the model and rifle are parallel, and the position shows an offset of 5cm along the sensor's  $\hat{x}$  axis and 10cm along its  $-\hat{z}$  axis. This result is included in the configuration file in the section containing the various rifle parameters and labeled as `rs2r` as shown in Appendix A.

The matrix,  ${}^{uw}\mathbf{H}_{ts}$ , specifies the torso sensor's position and orientation with respect to Jack's UPWAIST matrix. Although UPWAIST and shoulder coordinate systems are not the same, their axes are parallel and Eqs (27), (29) and (30) can be also used in the computation of this matrix where the orientation of the UPWAIST coordinate system is determined with respect to the transmitter as follows:

$${}^{uw}\mathbf{R}_{ts} = \left[ \begin{array}{ccc} {}^{tx}\hat{\mathbf{y}}_{sh} & {}^{tx}\hat{\mathbf{x}}_{sh} & -{}^{tx}\hat{\mathbf{z}}_{sh} \end{array} \right]^T {}^{tx}\mathbf{R}_{ts}^{side}, \quad (36)$$

where  ${}^{tx}\mathbf{R}_{ts}^{side}$  is the orientation of the torso sensor taken during the arm calibration. Finally, the homogeneous transformation matrix is computed as follows:

$${}^{uw}\mathbf{H}_{ts} = \left[ \begin{array}{ccc|c} {}^{uw}\mathbf{R}_{ts} & -0.040 & -0.015 & 0.0 \\ \hline 0 & 0 & 0 & 1 \end{array} \right], \quad (37)$$

Where the position vector is an assumption made about the approximate position of the torso sensor with respect to the UPWAIST coordinate system. **IMPORTANT: if the torso sensor were to be worn in a significantly different position this vector would have to be changed in the code and recompiled.**

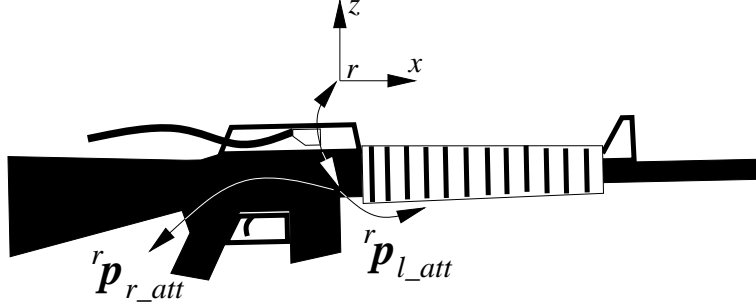


Figure 12: Arm attachment positions for performing snap-to.

### B. Snap-To

Due to assumptions made about the Jack model, approximations used in the calibration procedure and sensor errors, the hands of the Jack model will not be in the proper position with respect to the rifle model when the real hands grasp the real rifle. To solve this problem, a “snap-to” algorithm has also been implemented where the graphical rifle snaps to Jack’s hands when the real hands approach within a certain distance of the appropriate positions on the rifle. These positions are called the right and left hand attachment points,  $^r p_{r\_att}$  and  $^r p_{l\_att}$ , as illustrated in Figure 12 and are specified in the configuration file that is contained in Appendix A.

The attachment points are computed with respect to the transmitter using a calibration matrix and the rifle sensor data as follows:

$$^{tx} p_{att} = ^{tx} \mathbf{H}_{rs} {}^{rs} \mathbf{H}_r {}^r p_{att}, \quad (38)$$

where the *att* subscript corresponds to either the right (*r\_att*) or left (*l\_att*) attachment point. The distance between this point and the corresponding wrist sensor is then determined.

If either (but not both) distance is below a certain threshold (currently 25cm) the position of the rifle is modified so that the particular attachment point is equal to the sensor position as follows (for the right hand in this example):

$$^{uw} p_r = ^{uw} p_{rw} - ^{uw} \mathbf{R}_r {}^r p_{r\_att} \quad (39)$$

where  $^{uw} p_{rw}$  is the position of the right wrist of the Jack model with respect to the UPWAIST coordinate system (this is extracted from the Jack model after the joint angles that were computed in the inverse kinematics algorithm are applied). Also,  $^{uw} \mathbf{R}_r$  is the rotation portion of the matrix computed in Eq. (34) and  $^{uw} p_r$  computed here replaces the position vector of the same matrix. The same procedure can be done for the left hand. By modifying the position vector only, the orientation of the rifle that is sensed can be preserved.

In the case where both wrist sensor are within there respective threshold distances a slightly different approach must be taken because this is a closed-loop configuration where the sensor data conflicts with the kinematic constraint equations. Two approaches can be taken. The first is snap the rifle to the right hand and command a new set of joint angles to

snap the left hand to its attachment position. While this would maintain the orientation of the rifle, the added computational burden of recomputing the inverse kinematics for the left arm is significant, and the NPSNet computation does not easily allow for additional changes to the Jack model. Therefore, a second approach was taken where the rifle is snapped to the right hand and the rifle's orientation is then modified to come "close" to lining up with the left hand.

This change in orientation is accomplished by computing a new rotation matrix,  ${}^{uw}\mathbf{R}_r$ . First, the  $x$ -axis of the rifle set parallel to vector from the right hand to the left hand positions taken from the Jack model:

$${}^{uw}\hat{\mathbf{x}}_r = \frac{{}^{uw}\mathbf{p}_{lw} - {}^{uw}\mathbf{p}_{rw}}{\|{}^{uw}\mathbf{p}_{lw} - {}^{uw}\mathbf{p}_{rw}\|}, \quad (40)$$

and  $z$ -axis is left unchanged for now, that is,  ${}^{uw}\hat{\mathbf{x}}_r$  equals the third column of  ${}^{uw}\mathbf{R}_r$  as it was computed in Eq. (34). This will maintain the same general roll orientation of the rifle when the  $y$ -axis is computed as follows:

$${}^{uw}\hat{\mathbf{y}}_r = \frac{{}^{uw}\hat{\mathbf{z}}_r \times {}^{uw}\hat{\mathbf{x}}_r}{\|{}^{uw}\hat{\mathbf{z}}_r \times {}^{uw}\hat{\mathbf{x}}_r\|}. \quad (41)$$

Finally, the  $z$ -axis is orthogonalized:

$${}^{uw}\hat{\mathbf{z}}_r = \frac{{}^{uw}\hat{\mathbf{x}}_r \times {}^{uw}\hat{\mathbf{y}}_r}{\|{}^{uw}\hat{\mathbf{x}}_r \times {}^{uw}\hat{\mathbf{y}}_r\|}, \quad (42)$$

and the new rotation matrix is constructed:

$${}^{uw}\mathbf{R}_r = \begin{bmatrix} {}^{uw}\hat{\mathbf{x}}_r & {}^{uw}\hat{\mathbf{y}}_r & {}^{uw}\hat{\mathbf{z}}_r \end{bmatrix}. \quad (43)$$

It this new rotation matrix and position vector that are used in the computation of the **pfCoord** described above. Note, that the assumption that the  $x$ -axis lies on the line between the two hand positions is crude and results in the rifle stock floating a few inches above the left hand in the graphical model when snap-to to both hands is attempted. This is an area requiring further development.

### C. Targeting

The final task attempted at AUSA '95 was the use of the rifle sensor to perform targeting within the walk-in synthetic environment (WISE). This is actually an extremely difficult problem that requires calibration of the projection screens with respect to the Polhemus transmitter. Lacking this only a crude targeting scheme could be implemented in which the aiming only approximately follows the direction of the rifle.

Since NPSNet performs targeting under the assumption that the  $y$  axis determines the aim, an additional calibration matrix is introduced which performs the transformation from the rifle model's coordinate system (with the  $x$ -axis along the barrel) to the NPSNet's

coordinate system (with the  $y$ -axis along the barrel):

$${}^r\mathbf{R}_{aim} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \quad (44)$$

The orientation of orientation of the aiming coordinate system is computed as follows:

$${}^{hull}\mathbf{R}_{aim} = {}^{hull}\mathbf{R}_{tx} {}^{tx}\mathbf{R}_{rs} {}^r\mathbf{R}_{aim}, \quad (45)$$

where  ${}^{hull}\mathbf{R}_{tx}$  is the orientation of the transmitter with respect to the “*hull*” coordinate system that is used to define the orientation of any entity’s coordinate system with respect to Performer’s global (terrain-fixed) coordinate system. In the current version of the code, this matrix is set to the identity matrix, which means the  $y$ -axis of the transmitter **MUST** be aligned with the forward direction of the human and the  $z$ -axis is up. The position of the rifle with respect to this hull coordinate system is constant at 1.5m along the  $z$ -axis. This implementation is crude and falls in an area requiring a great deal of research and development before a proper approach is developed.

## 7. HMD Tracking: `hmdClass`

Using the new Fastrak device driver, the old HMD tracking code in NPSNet was replaced with a much more efficient, low-latency version. Because this requires one of the Polhemus sensor ports, it cannot currently be used in conjunction with the upper body tracking system in its present state. This setup requires that the HMD’s Polhemus sensor be plugged into Port One – the same as the torso sensor.

### A. Transmitter Alignment

As it is configured now, the wearer of the HMD stands or sits at a table containing a pair Flight Control Sticks (FCS’s) as shown in Figure 13. The “front” of the human in the virtual environment is always towards the table in the real world. The virtual human’s heading can be changed using the FCS controls and the user can orient the view with respect to this heading by rotating the HMD with respect to the table. As a result, the **hull** dynamic coordinate system (DCS) that is associated with the Jack graphical model in the NPSNet virtual environment can be associated with a static coordinate system with respect to the real environment. This coordinate system is shown on the table in Figure 13.

The Fastrak device driver will return the orientation of the HMD’s sensor with respect to the transmitter. To reduce the amount of calculation that must be performed by the application, the transmitter is aligned to the coordinate system just described. The proper alignment parameters to place in the configuration file can be determined by running a utility as follows from the `npsnet` directory:

```
% ./bin/fastrak_test ./datafiles/fastrak_test.dat
```

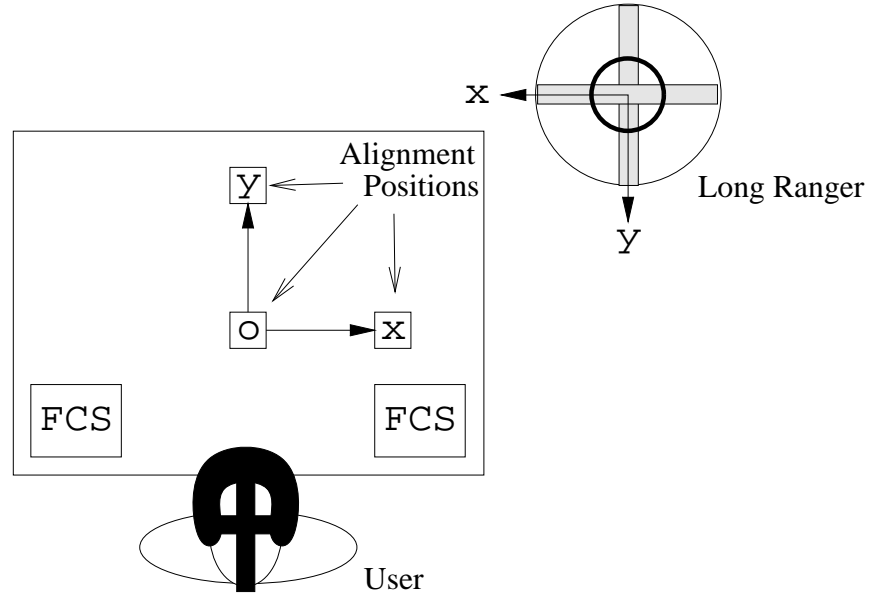


Figure 13: Top view of surface containing the human's flight control sticks (FCS's). The alignment positions also correspond to the axes of the hull coordinate system associated with every vehicle in NPSNet where  $y$  always points forward and  $x$  points to the right. The Long Ranger's coordinate system corresponds to its default coordinate system as it is mounted in the laboratory now.

It will initialize the Fastrak unit and prompt the user to place a sensor connected to port one in three different positions corresponding to the origin, a point along the  $x$ -axis, and a point along the  $y$ -axis as indicated by the small boxes on the tabletop in Figure 13.

The program outputs the alignment information that should be placed in the appropriate datafile in NPSNet. This file is specified in the `config/config.vim_new` configuration file after the new `HMD_FASTRAK_FILE` parameter (currently `datafiles/fastrak_vim.dat`). A transmitter orientation like the one illustrated in Figure 13 could result in the following alignment information:

```
STATION1_PARAM:
hemisphere:    0  0 -1
origin:        0.22004    0.09843    -1.59384
x_point:       -0.997687    0.00264599 -0.0679245
y_point:       0.000388254 -0.999004    -0.0446189
```

This information only needs to be updated when the forward direction of the human in the real world needs to be changed. Note that specific (non-zero) origin information is not necessary because the HMD tracking uses orientation and not position of the sensor.

### B. HMD Calibration and Use

Once the transmitter is aligned, the NPSNet application can be started. The application instantiates an object of the `hmdClass` which reads the appropriate `config` file, and in turn, instantiates an object of the `FastrakClass`. The next step during construction of the `hmdClass` object is to call the class's `calibrate()` function to boresight the HMD.

The boresighting procedure is accomplished by placing the HMD in a predefined orientation (facing the direction that will be used as 'forward' from then on) and sampling the sensor. This measures the orientation of the HMD's internally mounted sensor with respect to some coordinate system that is associated with the human's head (or camera) in the virtual environment. If this orientation is represented by the  $3 \times 3$  rotation matrix  ${}^{rx}\mathbf{R}_{hmd}$ , then it may be computed as follows:

$${}^{rx}\mathbf{R}_{hmd} = {}^{tx}\mathbf{R}_{rx}^T {}^{tx}\mathbf{R}_{hmd}^{cal} \quad (46)$$

where  $rx$  and  $tx$  refer to the receiver (sensor) and transmitter coordinate systems (as before), and  $hmd$  refers to the HMD or camera coordinate system that specifies the view direction in the virtual environment. The rotation matrix,  ${}^{tx}\mathbf{R}_{hmd}^{cal}$ , is the calibration matrix corresponding to the predefined HMD orientation with respect to the aligned transmitter coordinate system, and  ${}^{tx}\mathbf{R}_{rx}^T$  is the matrix returned by the `FastrakClass` that specifies the orientation of the sensor with respect to the transmitter. In this case the HMD is placed or worn with the person view pointing straight along the  $y$ -axis. Since the  $y$ -axis is the forward-looking vector, this calibration matrix is the identity matrix. Note that `calibrate()` may be called anytime during the NPSNet application to reboresight the HMD.

During normal operation of the simulator, the NPSNet application queries the `hmdClass` object for the current HMD orientation with respect to the hull coordinate system,  ${}^{hull}\mathbf{R}_{hmd}$ . The `hmdClass` object queries the `FastrakClass` object for the current sensor orientation  ${}^{tx}\mathbf{R}_{rx}^T$  and performs the follow operation to obtain the desired result:

$${}^{hull}\mathbf{R}_{hmd} = {}^{tx}\mathbf{R}_{rx} {}^{rx}\mathbf{R}_{hmd} \quad (47)$$

Because the transmitter has been aligned to the hull using the configuration parameters, the superscripts,  $tx$  and  $hull$  refer to the same coordinate system, and no additional matrix multiply is required.

The function `trak_view()`, in `view.cc`, combines the rotation matrix result with the `look.xyz` position to determine the complete homogeneous transformation specifying the view with respect to the hull. Multiplying this by an additional transformation using the `posture pfCoord` completes the specification of the view position and orientation with respect to virtual environments fixed coordinate system (often referred to as the Perform coordinates).

## 8. Summary and Future Work

This report contains a detailed description of the work that was completed to implement the Fastrak device driver; the algorithms to perform arm, rifle, and HMD tracking; and the

interface to NPSNet IV.9. There many areas that can be improved to make this a more generally usable system.

The first problem involves the baud rate of the Fastrak's serial connection. Limited to a 9600 baud serial connection, the maximum sample rate achieved was about 16 Hz when requesting both position and orientation information for four sensors. The reference manual [1] states the four sensors can be sampled as high as 30Hz which requires a higher baud rate. Attempts to increase the communication baud rate has been unsuccessful thus far. With 30Hz, some simple filtering could be implemented without introducing noticeable lag in NPSNet which runs at 15Hz. With the Ultratrak system that supports more sensors, the need for higher baud rates becomes even more important if the desired sample rates are to be obtained.

Part of the problem may be that the SGI serial ports are incapable of communicating above 9600 baud with software handshaking (XON/XOFF). In this case the solution may be to implement a serial cable capable of hardware handshaking. Another part of the problem may be that the `termio` software setup for the port is incorrect for the higher baud rates. Lack of time and the fact that the employees at Polhemus are of little help when dealing with SGI systems have both contributed to this continuing problem which deserves some study. A considerable amount of email (with code) from SGI/Polhemus users discussing potential solutions to this problem are available upon request.

The second area of potential improvement is in the implementation of the Fastrak device driver (the `FastrakClass`). The most immediate problem involves the initialization of the Fastrak unit. During instantiation of the class, a CTRL-Y character is sent to the unit which should trigger a 10 second reboot procedure (when the green light on the front of the unit flashes). Many times after power up the unit does not receive or recognize command. This situation requires that the software be restarted and/or the unit be power cycled in an attempt to reset the unit. This is not a foolproof procedure and may be required several times in various combinations.

This device driver code is based on a version developed by Paul Barham and Jiang Zhu in which the Fastrak device would provide sensor data only after it was specifically requested. This is referred to as the explicit polling method. Even when multiprocessing, this version would continually request data, wait, and receive it after the unit sampled the sensors. This was an extremely inefficient method and resulted in very low sample rates. The version developed here configures the Fastrak unit to continually sample the sensors and send the data over the serial line. It is continually monitored by the device driver which stores the data in a buffer for subsequent access by the application.

In the original approach, it was also simple to send new commands to reconfigure the unit. With the new approach, any configuration must be completed before the unit starts streaming sensor data. No support has been provided to suspend the function that reads this data so that new commands can be sent to the Fastrak unit. This is adequate for the applications discussed in this report since reconfiguration is not needed during operation, but it may limit its functionality for other applications.

Another limitation of the device driver is the ability to specify desired datatypes. Currently, the datatypes (e.g., position coordinates and euler angles) sent from the Fastrak unit must be the same for all sensors being sampled. This is specified as an optional param-

ter to the **FastrakClass** constructor and is a constraint imposed by this software and not the Fastrak unit itself. With the development of the full body sensor systems, support for heterogeneous data types across the sensors is anticipated. For example, orientation and position data would be required from one sensor representing the reference system while only orientation data would be required from sensors attached to each segment of the arm. This approach would significantly reduce the amount of information to be sent across the serial line. One possible solution, would be to encode the desired data types for each sensor and provide it with the sensor-specific data in the configuration file (the **STATIONx.PARAM** data presented on page 5), and modifying the constructor to read this data and configure the Fastrak unit appropriately.

Third, the **FastrakClass** provides no way of using the Fastrak unit’s commands to boresight the sensors (the functions exist, but they are not called in the constructor). This is partly due to the fact that the effect of these commands are not completely understood. It may provide a way to reduce some of the application’s computation by moving multiplications involving some of the constant calibration matrices to the Fastrak unit’s hardware.

With the development of the HMD software, inconsistent procedures for the use of the **FastrakClass** between this and the upper body system have been noticed. This suggests that some modifications to the system are needed. The first, is that configuration files for the Fastrak unit have different formats. The upper body system requires a single file with parameters that apply to the rifle as well as the Fastrak parameters, while the HMD file only contains the Fastrak parameters. It would probably be more consistent to have Fastrak configuration files that are separate from the application’s parameters; therefore, the upper body system would have a separate file containing the rifle parameters.

Second, the method of transmitter alignment is different for the HMD and upper body code. The upper body code assumes a certain amount of information about the orientation of the transmitter (that is, the  $z$ -axis is vertical). On the other hand, the HMD code requires some off-line sensor measurements to determine the actual transmitter orientation and produces a set of station parameters for the configuration file that will align the transmitter system as desired. The latter procedure could be incorporated into the upper body system to make it less sensitive to the actual transmitter orientation.

The use of calibration to measure the transformations between the torso and the shoulders is inconsistent with the use of the graphical models arm lengths rather than measuring the actual arms. Since the graphical model cannot be changed, it may be an improvement to measure the torso to shoulder transformation as it is stored in the Jack model. Then calibration would only be responsible for determining the position and orientation of the torso sensor with respect to Jack’s **UPWAIST** coordinate system.

Finally, a practice that makes the upper body algorithm very “brittle” is its use of constant calibration matrices such as the wrist sensor to wrist matrices,  $^{lws}\mathbf{H}_{lw}$  and  $^{rws}\mathbf{H}_{rw}$ , and the rifle sensor to rifle,  $^{rs}\mathbf{H}_r$ . It would be more general if these matrices could be measured during calibration (i.e., place the wrists and rifle in a known configuration and measure them). As a result the constraints on mounting the sensors would be much less strict. While the rotation portion could be easily obtained by placing the wrists/rifle in a known orientation and measuring the sensor orientation, the offset (position vector) can only be obtained by direct measurement of the sensor mount position relative to the corre-

sponding body's graphical coordinate system.

## 9. Acknowledgments

The author would like to acknowledge the help and guidance of Dr. David Pratt (PI), Paul Barham, and Randall Barker who were directly involved in the design and development of the dismounted infantry code in NPSNet. Thanks also go to the NPSNet sponsors: the U.S. Army Research Laboratories (ARL), Advanced Research Projects Agency (ARPA), Defense Modeling and Simulation Office (DMSO), U.S. Army Topographic Engineering Center (TEC), STRICOM, U.S. Army TRADOC Analysis Center (TRAC), and NPS Direct Funding.

## 10. References

- [1] Polhemus, Inc., Colchester, VT, *3SPACE FASTRAK User's Manual: Revision F*, November 1993.
- [2] J. Denavit and R. S. Hartenberg, "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," *ASME Journal of Applied Mathematics*, pp. 215–221, June 1955.
- [3] J. J. Craig, *Introduction to Robotics: Mechanics and Control*. Reading, MA: Addison-Wesley, 1986.
- [4] D. Tolani, "Inverse Kinematics of the Human Arm," HMS Laboratory Technical Report, University of Pennsylvania, March 1989.
- [5] W. H. Beyer, ed., *CRC Standard Mathematical Tables and Formulae*, (Boston), CRC Press, Inc, 1991.

### A. Configuration File Example

```
# This is a configuration file for the UpperbodyClass using four Polhemus
# sensors:  one on the rifle, one on each wrist, one on the upper torso
# (between the shoulder blades).
#
# The file format:
#   a). A line starting with a '#' is a comment line.
#   b). Each line must not contain more than 255 characters.
#   c). Maintain the order of the parameters (i.e., the station
#       parameters, hemisphere and alignment, must be the last
#       part of the file).
```

```

# ===== Parameters for the UpperBodyClass =====
# none

# ===== Parameters for the ArmClasses =====
# none

# ===== Parameters for the RifleClass =====
# rifle sensor to rifle model transformation and right and left hand
# attachment points with respect to the rifle model

## AR-15/M203 rubber rifle with grenade launcher
RIFLE:
rs2r:  1.0  0.0  0.0  0.051
      0.0  1.0  0.0  0.0
      0.0  0.0  1.0 -0.102
      0.0  0.0  0.0  1.0
ratt: -0.2288 -0.0127 -0.0505
latt:  0.1340  0.0254 -0.005

# ===== Parameters for the FastrakClass =====
# the serial port name for the FASTRAK
PORT: /dev/ttyd2

# Active sensors must be set to one here and set the switch on the front of
# the unit
WANTED_STATIONS: 1 1 1 1

# Parameters for the hemisphere and alignment of each station.
# Omitted stations will use the system defaults.
# The STATION#_PARAM line and the four parameter lines following it must
# immediately follow one another. There can be no comment lines among them.

# the hemisphere and alignment of station 1
STATION1_PARAM:
hemisphere: 0 0 -1
origin:     0 0 0
x_point:   -1 0 0
y_point:    0 -1 0

# the hemisphere and alignment of station 2
STATION2_PARAM:
hemisphere: 0 0 -1
origin:     0 0 0
x_point:   -1 0 0
y_point:    0 -1 0

# the hemisphere and alignment of station 3
STATION3_PARAM:

```

```
hemisphere: 0 0 -1
origin:      0 0 0
x_point:    -1 0 0
y_point:     0 -1 0

# the hemisphere and alignment of station 4
STATION4_PARAM:
hemisphere: 0 0 -1
origin:      0 0 0
x_point:    -1 0 0
y_point:     0 -1 0
```